Behavioral Cloning

Behavioral Cloning Project

The goals / steps of this project are the following:

- Use the simulator to collect data of good driving behavior
- Build, a convolution neural network in Keras that predicts steering angles from images
- Train and validate the model with a training and validation set
- Test that the model successfully drives around track one without leaving the road
- Summarize the results with a written report

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Files Submitted & Code Quality

1. Submission includes all required files and can be used to run the simulator in autonomous mode

My project includes the following files:

- model.py containing the script to create and train the model
- drive.py for driving the car in autonomous mode
- model.h5 containing a trained convolution neural network
- writeup_report.md or writeup_report.pdf summarizing the results

2. Submission includes functional code

Using the Udacity provided simulator and drive.py file, the car can be driven autonomously around the track by executing

```
    $ python drive.py model.h5
```

3. Submission code is usable and readable

The model.py file contains the code for training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model, and it contains comments to explain how the code works. To train the model execute the following command:

```
    $ python model.py
```

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model consists of a convolution neural network with 5x5 and 3x3 filter sizes and depths between 24 and 64 (model.py lines 94-116)
The model includes RELU activation to introduce nonlinearity (The RELU activation function is been specified in the convolutional layers in model.py lines 101- 103), and the data is normalized in the model using a Keras lambda layer (code line 95).

2. Attempts to reduce overfitting in the model

The model contains downsample (subsample/strides = 2) in order to reduce overfitting (medel.py lines 101- 103).
The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 127). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 124).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving, recovering from the left and right sides of the road, counter clockwise driving, and the **sample driving data** provided in the class.

For details about how I created the training data, see the next section.

Model Architecture and Training Strategy

1. Solution Design Approach

The overall strategy for deriving a model architecture was to use a classification network and since this is a regression problem I just need single output node to directly predict the steering measurements so I won't apply an activation function like a softmax activation function to the output layer.

My first step was to use a convolution neural network model similar to the architecture published by the Autonomous Vehicle team at Nvidia, I thought this model might be appropriate because this is the network they used to train a real car to drive autonomously.
In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. I found that my first model had a low mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was overfitting.
To combat the overfitting, I modified the model by adding a subsample to the first three convolutional layers.

The final step was to run the simulator to see how well the car was driving around track one. There were a few spots where the vehicle fell off the

track, to improve the driving behavior in these cases, I collected more data specifically recovering from the left and right sides of the road. I also cleaned up the data set and removed all the bad examples where the car was heading toward the side of the road.
At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

The final model architecture (model.py lines 94- 116) consisted of a convolution neural network with the following layers and layer sizes:

Layer name	Description
Input	160 x 320 x 3 RGB image
Cropping	65 x 320 x 3 RGB image
Convolution 5x5	2x2 stride, valid padding, outputs 31, 158, 24
RELU	
Convolution 5x5	2x2 stride, valid padding, outputs 14, 77, 36
RELU	
Convolution 5x5	2x2 stride, valid padding, outputs 5, 37, 48
RELU	
Convolution 3x3	1x1 stride, valid padding, outputs 3, 35, 64
RELU	
Convolution 3x3	1x1 stride, valid padding, outputs 1, 33, 64
RELU	
Fully connected	Outputs 100
Fully connected	Outputs 50
Fully connected	Outputs 10
Fully connected	Outputs 1

Here is a visualization of the architecture

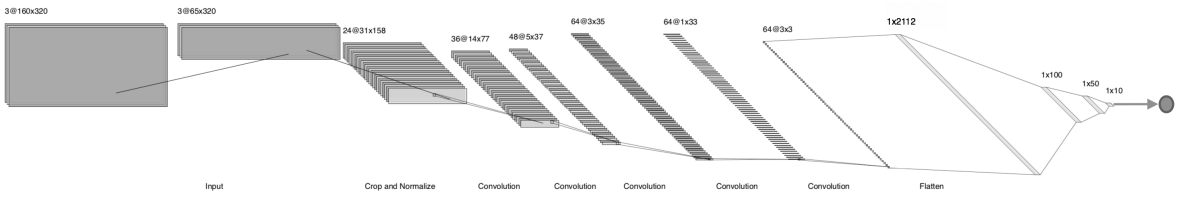

Figure 1: CNN Architecture

3. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded two laps on track one using center lane driving. Here is an example image of center lane driving:

I then recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn what to do if it gets off to the side of the road. These images show what a recovery looks like starting from right side of the road to the center:

Since track one has a left turn bias, I didn't want the data to be biased towards left turns, so I turn the car around and record counter-clockwise lap around the track. Here is a sample images from driving counter-clockwise:

I also, used the left and right camera images to train the model as if they were coming from the center camera. This way, I can teach my model how to steer if the car drifts off to the left or the right. To do that I created an adjusted steering measurements for the side camera images (model.py lines 40- 47). Images below show an example images captured from the right, left and center camera in order:

After the collection process, I had 95,373 number of data points. I then preprocessed this data by: first divide by 255 then subtract 0.5 to recenter the data. I also, cropped the images (model.py line 98). Here is an example of original image and cropped image:

Original image taken from the center camera of the simulator

Cropped image after passing through a Cropping2D layer

I finally randomly shuffled the data set and put 20% of the data into a validation set.
I used this training data for training the model. The validation set helped determine if the model was over or under fitting. The ideal number of epochs was 5 as evidenced by observing that the training loss and validation loss both decrease monotonically over all epochs.
I used an Adam optimizer so that manually training the learning rate wasn't necessary.